# ECS 235B, Lecture 14

February 8, 2019

# Trust Models

- Integrity models state conditions under which changes preserve a set of properties
  - So deal with the *preservation* of trustworthiness
- Trust models deal with confidence one can have in the initial values or settings
  - So deal with the *initial* evaluation of whether data can be trusted

# Definition of Trust

A *trusts* B if A believes, with a level of subjective probability, that B will perform a particular action, both before the action can be monitored (or independently of the capacity of being able to monitor it) and in a context in which it affects Anna’s own action.

- Includes subjective nature of trust
- Captures idea that trust comes from a belief in what we do not monitor
- Leads to transitivity of trust

# Transitivity of Trust

*Transitivity of trust*: if A trusts B and B trusts C, then A trusts C

- Not always; depends on A's assessment of B's judgment
- *Conditional transitivity of trust*: A trusts C when
  - B recommends C to A;
  - A trusts B's recommendations;
  - A can make judgments about B's recommendations; and
  - Based on B's recommendation, A may trust C less than B does
- *Direct trust*: A trusts C because of A's observations and interactions
- *Indirect trust*: A trusts C because A accepts B's recommendation

# Types of Beliefs Underlying Trust

- *Competence*: A believes B competent to aid A in reaching goal
- *Disposition*: A believes B will actually do what A needs to reach goal
- *Dependence*: A believes she needs what B will do, depends on what B will do, or it’s better to rely on B than not
- *Fulfillment*: A believes goal will be reached
- *Willingness*: A believes B has decided to do what A wants
- *Persistence*: A believes B will not change B’s mind before doing what A wants
- *Self-confidence*: A believes that B knows B can take the action A wants

# Evaluating Arguments about Trust (*con’t*)

- *Majority behavior*: A’s belief that most people from B’s community are trustworthy
- *Prudence*: Not trusting B poses unacceptable risk to A
- *Pragmatism*: A’s current interests best served by trusting B

# Trust Management

- Use a language to express relationships about trust, allowing us to reason about trust
  - Evaluation mechanisms take data, trust relationships and provide a measure of trust about the entity or whether an action should or should not be taken
- Two basic forms
  - Policy-based trust management
  - Reputation-based trust management

# Policy-Based Trust Management

- Credentials instantiate policy rules
  - Credentials are data, so they too may be input to the rules
  - Trusted third parties often vouch for credentials
- Policy rules expressed in a policy language
  - Different languages for different goals
  - Expressiveness of language determines the policies it can express

# Example: Keynote

- Basic units
  - Assertions: describe actions allowed to possessors of credentials
    - Policy: statements about policy
    - Credential: statements about credentials
  - Action environment: attributes describing action associated with credentials
- Evaluator: takes set of policy assertions, set of credentials, action environment and determines if proposed action is consistent with policy

# Example

- Consider email domain: policy assertion authorizes holder of mastercred for all actions:

```
Authorizer: "POLICY"
Licensees: "mastercred"
```

- Credential assertion:

```
KeyNote-Version: 2
Local-Constants: Alice="cred1234", Bob="credABCD"
Authorizer: "authcred"
Licensees: Alice || Bob
Conditions: (app_domain == "RFC822-EMAIL") &&
              (address ~= "^.*@keynote\\.ucdavis\\.edu$")
Signature: "signed"
```

- Compliance Value Set: { "_MIN_TRUST", "_MAX_TRUST" }

# Example: Results

- Evaluator given action environment:

```
 _ACTION_AUTHORIZERS=Alice
app_domain = "RFC822-EMAIL"
address = "snoopy@keynote.ucdavis.edu"
```

  it satisfies policy, so returns _MAX_TRUST

- Evaluator given action environment:

```
 _ACTION_AUTHORIZERS=Bob
app_domain = "RFC822-EMAIL"
address = ”opus@admin.ucdavis.edu"
```

  it does not satisfy policy, so returns _MIN_TRUST

# Example 2

- Consider separation of duty: policy assertion delegates authority to pay invoices to entity with credential “fundmgrcred”:

```
Authorizer: "POLICY"
Licensee: "fundmgecred"
Conditions: (app_domain == "INVOICE" && @dollars < 10000)
```

- Credential assertion (requires 2 signatures on any expenditure:

```
KeyNote-Version: 2
Comment: This credential specifies a spending policy
Authorizer: "authcred"
Licensees: 2-of("cred1", "cred2", "cred3", "cred4", "cred5")
Conditions: (app_domain=="INVOICE")      # note nested clauses
                -> { (@dollars) < 2500) -> "Approve";
                     (@dollars < 7500) -> "ApproveAndLog"; };
Signature: "signed"
```

- Compliance Value Set: { “Reject”, “ApproveAndLog”, “Approve” }

# Example 2: Results

- Evaluator given action environment:

```
_ACTION_AUTHORIZERS = "cred1,cred4"
app_domain = "INVOICE"
dollars = "1000"
```

  it satisfies first clause of condition, and so policy, so returns Approve

- Evaluator given action environment:

```
_ACTION_AUTHORIZERS = "cred1"
app_domain = "INVOICE"
dollars = "1500"
```

  it does not satisfy policy as too few Licensees, so returns Reject

# Example 2: Results

- Evaluator given action environment:

```
_ACTION_AUTHORIZERS = "cred1,cred2"
app_domain = "INVOICE"
dollars = "3541"
```

  it satisfies second clause of condition, and so policy, so returns ApproveAndLog

- Evaluator given action environment:

```
_ACTION_AUTHORIZERS = "cred1,cred5"
app_domain = "INVOICE"
dollars = "8000"
```

  it does not satisfy policy as amount too large, so returns Reject

# Reputation-Based Trust Management

- Use past behavior, information from other sources, to determine whether to trust an entity
- Some models distinguish between direct, indirect trust
- Trust category, trust values, agent's identification form *reputation*
- *Recommendation* is trust information containing at least 1 reputation
- Systems use many different types of metrics
  - Statistical models
  - Belief models (probabilities may not sum to 1, due to uncertainty in belief)
  - Fuzzy models (reasoning involves degrees of trustworthiness)

# Example 1

- Direct trust: –1 (untrustworthy), 1 to 4 (degrees of trust, increasing), 0 (canot make trust judgment)
- Indirect trust: –1, 0 (same as for direct trust), 1 to 4 (how close the judgment of recommender is to the entity being recommended to)
- Formula: $t(T, P) = tv(T)\prod_{i=1}^{n} \frac{tv(R_i)}{4}$ where *T* is entity of concern, *P* trust path, *tv*(*x*) trust value of *x*, *t*(*T*,*P*) overall trust in T based on trust path *P*

# Example 1

- Amy wants Boris' recommendation about Danny so she asks him
  - Amy trusts Boris' recommendations with trust value 2 as his judgment is somewhat close to hers
- Boris doesn't know Danny, so he asks Carole
  - He trusts her recommendations with trust value 3
- Carole believes Danny is above average programmer, so she replies with a recommendation of 3
- Boris adds this to the end of the recommendation
- Path is (Amy—Boris—Carole—Danny), so R1 = Boris, R2 = Carole, *T* = Danny, and

$$T(\text{"Danny"}, P) = 3 \times \frac{2}{4} \times \frac{3}{4} = 1.125$$

# Example 2

- PeerTrust uses metric based on complaints
- *u*
- *P* is a node in a peer-to-peer network
- *p*(u, t) in P is node that u interacts with in transaction t
- S(u,t) amount of satisfaction u gets from p(u,t)
- I(u) total number of transactions
- Trust value of *u*: T(u) = $\sum_{t=1}^{I(u)} S(u,t)Cr(p(u,t))$
- Credibility of node x's feedback: Cr(x) = $\sum_{t=1}^{I(x)} S(x,t) \frac{T(p(x,t))}{\sum_{y=1} I(x) T(p(x,y))}$
- So credibility of *x* depends on prior trust values

# Key Points

- Integrity policies deal with trust
  - As trust is hard to quantify, these policies are hard to evaluate completely
  - Look for assumptions and trusted users to find possible weak points in their implementation
- Biba, Lipner based on multilevel integrity
- Clark-Wilson focuses on separation of duty and transactions

# Availability

- Goals
- Deadlock
- Denial of service
    - Constraint-based model
    - State-based model
- Networks and flooding
- Amplification attacks

# Goals

- Ensure a resource can be accessed in a timely fashion
  - Called “quality of service”
  - “Timely fashion” depends on nature of resource, the goals of using it
- Closely related to safety and liveness
  - Safety: resource does not perform correctly the functions that client is expecting
  - Liveness: resource cannot be accessed

# Key Difference

- Mechanisms to support availability in general
  - Lack of availability assumes average case, follows a statistical model
- Mechanisms to support availability as security requirement
  - Lack of availability assumes worst case, adversary deliberately makes resource unavailable
  - Failures are non-random, may not conform to any useful statistical model

# Deadlock

- A state in which some set of processes block each waiting for another process in set to take come action
  - *Mutual exclusion*: resource not shared
  - *Hold and wait*: process must hold resource and block, waiting other needed resources to become available
  - *No preemption*: resource being held cannot be released
  - *Circular wait*: set of entities holding resources such that each process waiting for another process in set to release resources
- Usually not due to an attack

# Approaches to Solving Deadlocks

- *Prevention*: prevent 1 of the 4 conditions from holding
  - Do not acquire resources until all needed ones are available
  - When needing a new resource, release all held
- *Avoidance*: ensure process stays in state where deadlock cannot occur
  - *Safe state*: deadlock can not occur
  - *Unsafe state*: may lead to state in which deadlock can occur
- *Detection*: allow deadlocks to occur, but detect and recover

# Denial of Service

- Occurs when a group of authorized users of a service make that service unavailable to a (disjoint) group of authorized users for a period of time exceeding a defined maximum waiting time
  - First “group of authorized users” here is group of users with access to service, whether or not the security policy grants them access
  - Often abbreviated “DoS” or “DOS”
- Assumes that, in the absence of other processes, there are enough resources
  - Otherwise problem is not solvable unless more resources created
  - Inadequate resources is another type of problem

# Components of DoS Model

- *Waiting time policy*: controls the time between a process requesting a resource and being allocated that resource
  - Denial of service occurs when this waiting time exceeded
  - Amount of time depends on environment, goals
- *User agreement*: establishes constraints that process must meet in order to access resource
  - Here, “user” means a process
  - These ensure a process will receive service within the waiting time

# Constraint-Based Model (Yu-Gligor)

- Framed in terms of users accessing a server for some services
- *User agreement*: describes properties that users of servers must meet
- *Finite waiting time policy*: ensures no user is excluded from using resource

# User Agreement

- Set of constraints designed to prevent denial of service
- $S_{seq}$ sequence of all possible invocations of a service
- $U_{seq}$ set of sequences of all possible invocations by a user
- $U_{Ii,seq} \subseteq U_{seq}$ that user $U_i$ can invoke
  - $C$ set of operations $U_i$ can perform to consume service
  - $P$ set of operations to produce service user $U_i$ consumes
  - $p < c$ means operation $p \in P$ must precede operation $c \in C$
  - $A_i$ set of operations allowed for user $U_i$
  - $R_i$ set of relations between every pair of allowed operations for $U_i$

# Example

Mutually exclusive resource

- $C$ = { *acquire* }
- $P$ = { *release* }
- For $p_1$, $p_2$, $A_i = \{ acquire_i, release_i \}$ for $i = 1, 2$
- For $p_1$, $p_2$, $R_i = \{ ( acquire_i < release_i ) \}$ for $i = 1, 2$

# Sequences of Operations

- $U_i(k)$ initial subsequence of $U_i$ of length k
  - $n_o(U_i(k))$ number of times operation $o$ occurs in $U_i(k)$
- $U_i(k)$ safe if the following 2 conditions hold:
  - if $o \in U_{i,seq}$, then $o \in A_i$; and
    - That is, if $U_i$ executes $o$, it must be an allowed operation for $U_i$
  - for all $k$, if $(o < o') \in R_i$, then $n_o(U_i(k)) \geq n_{o'}(U_i(k))$
    - That is, if one operation precedes another, the first one must occur more times than the second

# Resources of Services

- $s \in S_{seq}$ possible sequence of invocations of services
- $s$ blocks on condition $c$
  - May be waiting forservice to become available, or processing some response, etc.
- $o_i^*(c)$ represents operation $o_i$ blocked, waiting for $c$ to become true
  - When execution results, $o_i(c)$ represents operation
  - Note that when $c$ becomes true, $o_i^*(c)$ may not resume immediately

# Resources of Services

- $s(0)$ initial subsequence of $s$ up to operation $o_i^*(c)$
- $s(k)$ subsequence of operations between $k$-1st, $k$th time $c$ becomes true after $o_i^*(c)$
- $o_i^*(c) \rightarrow^{s(k)} o_i(c)$: $o_i$ blocks waiting on $c$ at end of $s(0)$, resumes operation at end of $s(k)$
- $S_{seq}$ *live* if for every $o_i^*(c)$ there is a set of subsequences $s(0), ..., s(k)$ such that it is initial subsequence of some $s \in S_{seq}$ and $o_i^*(c) \rightarrow^{s(k)} o_i(c)$

# Example

- Mutually exclusive resource; consider sequence

  $$( acquire_i, release_i, acquire_i, acquire_i, release_i )$$

  with $acquire_i$, $release_i \in A_i$, $(acquire_i, release_i) \in R_i$; $o = acquire_i$, $o' = release_i$
- $U_i(1) = (acquire_i) \Rightarrow n_o(U_i(1)) = 1, n_{o'}(U_i(1)) = 0$
- $U_i(2) = (acquire_i, release_i) \Rightarrow n_o(U_i(2)) = 1, n_{o'}(U_i(2)) = 1$
- $U_i(3) = (acquire_i, release_i, acquire_i) \Rightarrow n_o(U_i(3)) = 2, n_{o'}(U_i(3)) = 1$
- $U_i(4) = (acquire_i, release_i, acquire_i, acquire_i) \Rightarrow$

  $n_o(U_i(4)) = 3, n_{o'}(U_i(4)) = 1$
- $U_i(5) = (acquire_i, release_i, acquire_i, acquire_i, release_i) \Rightarrow$

  $n_o(U_i(5)) = 3, n_{o'}(U_i(5)) = 2$
- As $n_o(U_i(k)) > n_{o'}(U_i(k))$ for $k = 1, ..., 5$, the sequence is safe

# Example (*con’t*)

- Let $c$ be true whenever resource can be released
  - That is, initially and whenever a $release_i$ operation is performed
- Consider sequence: ($acquire_1$, $acquire_2{}^*(c)$, $release_1$, $release_2$, ... , $acquire_k$, $acquire_{k+1}(c)$, $release_k$, $release_{k+1}$, ...)
- For all $k \geq 1$, $acquire_i{}^*(c) \rightarrow^{s(1)} acquire_{k+1}(c)$, so this is live sequence
  - Here, $acquire_{k+1}(c)$ occurs between $release_k$ and $release_{k+1}$

# Expressing User Agreements

- Use temporal logics
- Symbols
  - $\Box$: henceforth (the predicate is true and will remain true)
  - $\Diamond$: eventually (the predicate is either true now, or will become true in the future)
  - $\rightsquigarrow$: will  lead to (if the first part is true, the second part will eventually become true); so $A \rightsquigarrow B$ is shorthand for $A \Rightarrow \Diamond B$

# Example

- Acquiring and releasing mutually exclusive resource type
- User agreement: once a process is blocked on an *acquire* operation, enough *release* operations will release enough resources of that type to allow blocked process to proceed

**service** resource_allocator

**User agreement**

$$in(acquire) \leadsto ((\Box\Diamond(\#active\_release > 0) \vee (free \geq acquire.n))$$

- When a process issues an *acquire* request, at some later time at least 1 *release* operation occurs, and enough resources will be freed for the requesting process to acquire the needed resources